1984

A state-driven vpe system

Peter Strazdins
University of Wollongong

THE UNIVERSITY OF WOLLONGONG

DEPARTMENT OF COMPUTING SCIENCE

A STATE-DRIVEN VPE SYSTEM

Peter Strazdins

Department of Computing Science
University of Wollongong

Submitted in partial fulfillment of the requirements of CSCI321
Software Project at the University of Wollongong in 1983.

Preprint No 84-9 **May 31, 1984**

A STATE-DRIVEN VPE SYSTEM.

Peter Strazdins.

CSCI321 Software Project.
Department Of Computing Science.
University Of Wollongong.

TABLE OF CONTENTS FOR THIS DOCUMENT.

1. PROJECT COMMENCEMENT.
1.1. Introduction.
1.2. Definitions.
1.3. The Existing VPE System.

2. PROJECT SPECIFICATIONS.
2.1 Overview.
2.2 Specifications Of The State-Driven VPE System.
2.3 Project Goals.

3. SYSTEM DESIGN.
3.1. Major Sections Of The System.
3.2. System Block Diagrams.
3.3. Software For A State-Driven VPE System.
3.4. User Interface Of The VPE System.

4. DETAILED DESIGN - THE VPE GENERAL PROCEDURES.
4.1. Overview.
4.2. Data Structures For The General Procedures.
4.3. The Data Structure Initialization Procedures.
4.4. The 'Universal' General Procedure.
4.5. The VPE Screen Procedures.

5. THE DESIGN OF THE VPE PARSER.
5.1. Acceptable Displayed Programs For The VPE Parser.
5.2. Handling Of Input Errors.
5.3. Tasks Of The VPE Parser.
5.4. Code Generation.

6. PROJECT IMPLEMENTATION.
6.1. Implementation Of The Data Structure Procedures.
6.2. Implementation Of The VPE Screen Procedures.
6.3. Implementation Of VPE Output Programs.

7. PREPARATION OF LESSONS FOR THE STATE-DRIVEN SYSTEM.

8. EXTENSIONS FOR THE STATE-DRIVEN SYSTEM.
8.1. Addition Of A New Data Type.
8.2 Addition Of Internal Procedures/ Functions.

9. CONCLUSION.

ACKNOWLEDGEMENTS.

BIBLIOGRAPHY.

APPENDIX A.

1. PROJECT COMMENCEMENT.

1.1. Introduction.

This is a specification for a 'state-driven' VISIBLE PROGRAM EXECUTION (VPE) system which, given a small Pascal 'displayed program', automatically produces a VPE 'output program'. The VPE 'output program', when executed, visibly executes the 'displayed program' on a computer terminal.

A VPE system has already been developed at the University of Wollongong. In this system, for every executable statement in the Pascal 'displayed program', there are several procedure calls to manipulate the screen accordingly. This approach leads to complexities in implementation which makes the system difficult to extend. A new approach is proposed to be implemented by this project. It is based on the concept of 'screen states' of the 'displayed program', and uses generalized data structures. This approach is expected to make the proposed system more versatile and extensible, and enable some fundamentally new features in Visual Program Execution.

1.2. Definitions.

1.2.1. VPE Displayed Program.

A small, compilable Pascal program suitable for VPE. Alternately, it may be a self-contained, compilable Pascal procedure/function suitable for VPE.
This program is submitted to the VPE system to be visibly executed.

1.2.2. VPE Output Program.

A (Pascal) program that visibly executes the VPE Displayed Program, and is the output of the VPE system.

1.2.3. VPE Lesson.

This is the process of executing a VPE Output Program. The result is a 'lesson' displayed on a terminal.

1.2.4. Screen States.

The screen states of a (Pascal) VPE Displayed Program are defined as follows:

i. For every entry into Pascal block there is a screen state; similarly for every exit from a block there is also a screen state.

ii. For every Pascal assignment statement there is also a screen state.

iii. For every condition evaluation inside a Pascal *if*, *while* or *repeat* statement, there is a screen state. *Similarly*, *for every for* statement index modification inside a Pascal *for* statement, there is also screen state.

Each state has also a depth attribute, which corresponds to the semantic 'depth' of the corresponding code of the VPE Displayed Program. This definition gives a tree-like hierarchy of screen states for VPE Displayed Programs of similar structure to its Pascal semantic tree. An inherent property of these screen states, relevant to VPE, is its *level* in this hierarchy. An example of the screen states of a selection sort VPE Displayed Program will be given later. Henceforth, the screen states of a VPE Displayed Program may be simply referred to as states.

1.2.5. VPE Preparer.

This is the person, who, given a VPE Displayed Program for visible execution, must prepare, with the assistance of the software in the VPE system, the corresponding VPE Output program.

1.2.6. VPE User.

This refers to the person (ie. student) executing the VPE Output Program.

1.2.7. Screen Layout.

This is what the VPE User sees on the terminal. It includes:

i. The (Pascal) code of the VPE Displayed Program.

ii. Boxes displaying the current values of the variables/conditions in the VPE Displayed Program.

iii. A boxed arrow pointing to the next executable statement in the VPE Displayed Program.

1.2.8. VPE General Procedures.

This refers to a library of screen layout handling procedures called by the VPE Output Program. They form the backbone of a VPE system. They appear as #include files in the VPE Output Program.

1.2.9. VPE Parser.

This refers to the program that parses the VPE Displayed Programs and produces the corresponding VPE Output Program, with some interaction with the VPE Preparer. Its purpose is to make the VPE Preparer's task easier.

1.3. VPE System.

This is a large set of files, including:

i. A library of VPE Displayed Programs.

ii. A library of corresponding VPE Output Programs.

iii. A library of VPE General Procedures.

iv. An Interactive Screen Preparation (ISP) program.

v. VPE Parser program(s).

1.4. The Existing VPE System.

Before specifying this project, it will be necessary to briefly describe the existing VPE System:

The existing VPE System is implemented under the UNIX operating system. Three aspects of the system relevant to this project are the creation of the VPE Output Program, the structure of the VPE Output Program, and the execution of the VPE Output Program. Further information on the existing system may be found in reference [0].

1.4.1. The Creation Of The VPE Output Program.

Currently, the VPE Preparer, given a suitable VPE Displayed Program, must carry out the following procedure to produce a VPE Output Program:

i. Execute an ISP program which interactively assists the VPE Preparer to initialize the screen layout for the VPE Output Program.

ii. Execute the VPE Parser programs, which, with some interaction, and with the output of the ISP program, produce the VPE Output Program.

This may take up to 15 minutes, with most of the time taken to initialize the screen layout.

1.4.2. The Structure Of The VPE Output Program.

The VPE Output Program produced by the existing VPE Parser program contains:

i. Declarations of constants defining the screen layout. Typically, there are more than 30 of these.

ii. Declarations of VPE Displayed Program variables (under same identifiers as in the VPE Displayed Program) and declarations of variables used for screen layout book-keeping.

iii. #include directives to the appropriate files in the library of VPE General Procedures.

iv. A program body containing the Pascal statements from the VPE Displayed Program interspersed with (typically several) appropriate calls to VPE General Procedures (with typically long parameter lists) to manipulate the screen layout.

1.4.3. Execution Of The VPE Output Program.

The VPE Output Programs compile to object files of sizes typically from 10 to 20 K. Upon the VPE User executing such an object file:

i. Brief instructions are given , if required, on how to use the VPE System.

ii. The VPE User is asked which mode (query mode or normal mode) that he wishes to execute the program.

iii. The screen layout is initialized and visible execution is ready to begin at the first executable Pascal statement in the VPE Displayed Program.

iv. The VPE Output Program steps from one executable statement to the next executable statement of the VPE Displayed Program until it terminates. At each step, if in normal mode, any changes in the VPE Displayed Program's variables over the last step results in an appropriate modification of the screen layout; if in query mode, the VPE User is asked to supply the new values.

Those requiring more information on the user interface of the existing VPE System should execute the explain command on the University of Wollongong UNIX system or use the demonstration package provided on the system.

2. PROJECT SPECIFICATIONS

2.1. Overview.

The aim of this project is to design and implement a state-driven VPE System. A state-driven VPE system is a VPE System designed so that for every screen state of a VPE Displayed program, the corresponding VPE Output Program has some equivalent (ie. to the displayed code of that state) code. This is followed by calling a VPE General Procedure whose principal argument is a representation of the state of the VPE Displayed Program. A 'history' of the screen states and the values of the VPE Displayed Program variables are kept in such a system, allowing greater flexibility.

2.2. Specifications Of The State-Driven System.

The state-driven system to be implemented by this project has the following specifications:

i. accepts a restricted subset of VPE Displayed Programs which:

   a. contain the data types integer, nelements (a subset of an array[0..20] of integer), boolean, boolarray (a subset of an array[0..20] of boolean), char and chararray (a subset of an array[0..20] of char).

   b. contains the Pascal control constructs: if statements, while statements, for statements, and repeat statements.

c. contains no internally declared Pascal procedures or functions.

d. has the following restrictions on array indices:
the array indices must be expressions containing only integers, integer variables and Pascal integer operators.

e. is small enough to be displayed entirely on a 24*80 standard terminal screen.

Although this is a subset of that accepted by the existing VPE system, it is still general enough to have to solve the crucial screen layout manipulation problems a more general VPE system would have to solve. These restrictions are justifiable in the light that VPE is primarily intended to teach simple programming concepts.

ii. allows extensions (ie. inclusions of new data types to the system) to be an 'additive' update of the implemented system. ie. extension will mainly involve adding the particular screen layout manipulation procedures to the VPE General Procedures and modifying some well-documented modules in the implemented General Procedures.

iii. has its VPE Output Programs implemented in Pascal, as in the existing system, for portability reasons. (as the displayed language is Pascal).

iv. has the ability to visibly execute forward or backward, (ie. 'undo' computations), at the VPE User's request.

v. has the ability to visibly execute, at the VPE User's request, the displayed program at a higher level, (ie. compound statement level), rather than the simple 'line-by-line' mode available in present system.
ie. to visibly execute control constructs, procedures and compound statements in a single step.
This can give the VPE User a 'high level', as well as a 'low level' understanding of the displayed program.

The project will not fundamentally change the interface between the VPE User and the VPE System. Nor is it intended to fundamentally change the role of the VPE Preparer, and ISP will still be incorporated into the state-driven system as in the existing system. (although, ideally, ISP should take place in parallel with the VPE Parser). Hence, the project is intended to change fundamentally the form of the VPE Output Programs, making them simpler and more readable. Moreover, the VPE Output program will have, by a simpler interface with the VPE General Procedures, a closer correspondence with its VPE Displayed Program. (see selection sort example). This will involve extensive and non-trivial modifications to the VPE General Procedures and hence the VPE Parser.

The state-driven system should lead to a simpler implementation than the existing system, and should particularly simplify the implementation of the VPE Parser. This can be achieved by deferring until 'run-time' the screen layout manipulation decisions the VPE Parser has to make in the existing system. ie. the VPE General Procedures will make these decisions instead as the VPE Output Program is executed.

2.2.1. The VPE General Procedures for the state-driven system.

It is proposed that the VPE General Procedures assume the existence of the following data structures in every VPE Output Program:

i. A state table: a summary of all information (relevant to screen layout) of the screen states of the appropriate VPE Displayed Program.
The table may contain such information of the 'level' of the state, the VPE Displayed Program variables directly affected by execution of the code of the state, and the textual position (in the VPE Displayed Program) of the beginning of the state.

ii. A state stack: a history of the VPE Displayed Program screen states visibly executed so far during the execution of the VPE Output Program.

iii. A list of the VPE Displayed Program variables. This will contain such information as the way each variable is displayed on the screen layout, the type of each variable, and a history stack of all values assigned to each variable so far.

The following design policies are intended for the general procedures of the state-driven system:

i. To develop a powerful set of primitive screen layout manipulation procedures and primitive data structure manipulation procedures, which can easily be called from the higher level VPE General Procedures.

ii. To design the higher level general procedures to easily allow the inclusion of new primitive procedures. (ie. the inclusion of the primitive screen layout manipulation procedures for a new data type).

iii. To make the decisions of what screen layout manipulations are to be performed at each 'step' of the VPE Output Program, using the information in the above mentioned data structures. The general procedures are also to be responsible for initializing and maintaining these data structures.

iv. A 'universal' general procedure is to be designed, so that all VPE Output Programs can perform screen layout manipulations only through a call to this procedure. For every screen state of the VPE Displayed Program, there will appear, in the VPE Output Program, a single call to this procedure. It is proposed that this procedure needs only two arguments, namely the state number of the screen state that is to be visibly executed, and a list of the new values of the variables affected by the execution of the VPE Displayed Program code corresponding to that state.

2.2.2. The VPE Parser for the state-driven system.

The design of the VPE Parser will depend on the design of the VPE General Procedures and hence can only be fully realized once the design of the VPE Procedures is complete. It is proposed that the VPE Parser:

i. Be designed consistently with Pascal syntax and be easily extendible.

ii. Parse the VPE Displayed Program into its screen states, without necessarily checking for syntactic errors (which can be done by a Pascal compiler), and produce the entire VPE Output Program, with as little interaction with the VPE Preparer as possible. It is expected that this task will be considerably simplified by the proposed form of the VPE Output Program.

iii. Be designed to minimize the complexity of the algorithms involved, and so enhance the readability and maintainability of the VPE Parser.

2.3. Project Goals.

Since this project involves two dependent pieces of software, as the design of the VPE Parser cannot begin until the design of the VPE General Procedures is complete, it is proposed the project is done in two stages:

i. The design and almost complete implementation of the general procedures of the state-driven system; here, illustrative VPE Output Programs will be produced 'by hand' to test the design. ie. VPE Output Programs for a selection sort and an appropriate function calling program would probably be sufficient for this purpose.

ii. The design of the VPE Parser to automatically generate these output programs.

3. SYSTEM DESIGN.

3.1. Major Sections Of The System.

The system comprises of two largely orthogonal sections, which are treated separately in this design as follows:

i. The VPE General Procedures.
These are responsible for producing the VPE effects, as explained in the specifications for this project. They can be divided into two sections, the VPE Screen Procedures, and the VPE Data Structure Procedures. The VPE Screen Procedures are directly responsible for manipulating the screen layout, ie. setting boxes, arrows and getting the VPE User's responses, and are to be independent of the data structures used by the VPE Data Structure Procedures. The VPE Data Structure Procedures are responsible for maintaining the state table and its related data structures, and for calling the appropriate VPE Screen Procedures. The 'universal' general procedure, is the procedure principally called by the VPE Output Program, and makes the high-level decisions upon which other VPE Data Structure Procedures are called.

ii. The VPE Parser.
This is a 'Llama' compiler, which, given a VPE Displayed Program and its ISP screen layout record file, decomposes the displayed program into its screen states, and produce the corresponding VPE Output Program and its associated state table data file. The state table output file is read by the output program upon execution.

The VPE Parser is not expected to be implemented for this project. However, the design is given here as part of this project, because automatic generation of VPE Output Programs from VPE Source Programs is a very useful and convenient feature of a VPE system. The design of the parser is the motivation for doing the present project. Hence, this project would not be feasible if a parser could not be easily implemented for the proposed system.

3.2. System Block Diagrams.

i. The preparation cycle, ie. the preparation of a VPE lesson given the appropriate displayed program, is shown below. The project should make this cycle as easy as possible for the VPE Preparer.

```
VPE Displayed      -> ISP    -> screen layout
program 'display'               record file 'isp~~~~~'

'isp~~~~~',    -> VPE Parser -> VPE Output Program
'display'                       'display.p',
                                state table data file
                                'display.s'.
```

The compile cycle, in which the VPE Output program is compiled, is also relevant to system design:

```
'display.p',      -> Pascal compiler -> output program
                                        object file
#include files,                        'displayobj'
containing the
VPE General Procedures
and data structure
definitions.
```

The lesson cycle, in which a VPE User, executes the output program object file, is as follows:

```
'displayobj',  -> Pascal object code  -> VPE lesson
'display.s'       interpreter             on terminal
```

ii. The VPE General Procedures have a hierarchical structure expressed by the following diagrams. Below is the hierarchy for the VPE Data Structure Procedures:

VPE Data Structure Procedures.

data structure initialization procedures

'universal' general procedure.

data structure initialization procedures

state table initialization procedures

'program status' data structure initialization procedure

'universal' general procedure.

high-level 'screen adjusting' procedures.

data structure maintenance procedures

VPE Screen Procedure calling procedures.

data structure modification primitives

data structure retrieval primitives.

VPE Screen Procedures.

boolean/ condition procedures

integer procedures

character procedures

array procedures

general screen effect procedures

terminal dependent primitives

parameter initialization procedures.

3.3. Software For A State-driven VPE System.

A description of the files required is given in this section. These are of four types:

i. Files required for each VPE lesson.
These include a VPE displayed program, the corresponding (Pascal) VPE output program, state table data file and the object code file for the output program.

ii. Files generated by execution of a VPE lesson.
These include a debug file, used for tracing the execution of general procedures called during a lesson, and a file reporting the success of the VPE user, after completion of the lesson.

iii. #include files.
These appear in #include directives in the VPE Output programs, and include a definition of constants for the data structures used by the VPE General Procedures, a definition of these data structures, a declaration of the VPE variables referencing the data structures and the VPE General procedures. These constitute the main part of the object code of a VPE lesson, and are independent of any particular VPE displayed program.

iv. Files for preparing VPE lessons.
Firstly an ISP (Interactive Screen Preparation) program is required to assist the VPE preparer with the screen layout; the output of this program is in record format, suitable to be read by the VPE Parser; the information in the output will be entered in the state table the parser generates. Such a program is currently available. Secondly, a VPE Parser is required; it will have the following associated files:

(a) a Llama source file, containing the grammar, semantic actions and the required routines of the parser.

(b) the Pascal code of the source file.

(c) the object code of the source file.

3.4. User interface of the VPE system.

VPE is by nature highly interactive and hence the user interface is an important aspect of the system. The most important interface is between the VPE Screen Procedure and the VPE user. As this has already been defined for the existing VPE system, only the different aspects of this interface will be described here. The interface of the VPE Data Structure Procedures and the VPE Parser with the VPE preparer is less important, but will also be dealt with here.

i. VPE Screen Procedures.
These interact with the VPE user in essentially the same fashion as the existing system except in the following notable exceptions:

a. The VPE user, whenever prompted for input, can, by means of entering the VPE escape character (<ESC>) before pressing <RETURN>, request to change the mode of execution during a lesson. Upon doing this, the user will be prompted for control, and should respond as follows:
<optional sign :'+'or '-'> <optional integer> <optional 'f' or 'r'> <RETURN>
Entering a sign will only have an effect if an integer follows. Entering a '+' followed by an integer n will allow visible execution to pause at all VPE screen states whose depth is not more than depth of current state + n; similarly entering a '-' followed by an integer n will allow execution to henceforth pause in state whose depth is not more then depth of current state - n; entering an unsigned integer n will cause execution to pause at all states whose depth is not more then n. Entering an 'f' will force visible execution to go forwards; entering an 'r' will force visible execution to go backwards, if possible. ie. return to an earlier point in the lesson. The default is to leave visible execution as it is. The user will not be queried when execution is in reverse mode.

b. When the lesson does not halt during visible execution of a particular screen state, the variables affected are updated, as in the automatic mode of existing system, but the user is not prompted for any input and the statement counters are not displayed for such states.

c. When the user is prompted for the value of a displayed program variable, the last value assigned to that variable, if it exists, is displayed in inverse video on the right hand side of its box, which the user will overwrite with his response. This is necessary for accumulative statements such as 'x := x + 1' appearing in the displayed program.

d. When the user is prompted for the value of an array element, input will occur in the appropriate part of the box of the array, rather to a separate box as is done in the existing system. This simplifies screen layout and has conceptional advantages for VPE.

ii. The VPE Data Structure Procedures.
These interact with the preparer in the following manner:

a. Upon detection of an illegal condition in the data structures, the procedures will call a 'panic' routine, which displays a diagnostic message both to the terminal and to a debug file before halting execution. This testing of conditions provides a means of making assertions about these procedures.

b. During execution of all non-trivial procedures, the name of the procedure and any relevant arguments are written to the debug file, to enhance diagnosis of errors in these procedures.

iii. The VPE Parser.
As some information required in the state table cannot be deduced by the parser, it must ask the VPE preparer the following:

a. How each parameter (in the case of the displayed program being a Pascal procedure/function) is to be initialized. In particular, all 'value' parameters must be initialized, but 'name' parameters need not.

b. For each array appearing in the displayed program, the user is requested to provide the lower and upper (integer) bounds of the array.

c. Upon each assignment of an array element, the preparer will be asked whether the element is to be left in inverse video after the assignment.

d. For each index (integer expression) indexing an array in the displayed program, the preparer will be asked the 'offset', ie. how many lines underneath, of that index from the array. (an offset of 0 indicates the preparer does not wish the index to be displayed). This must be done by the parser as it is not determined by ISP.

4. DETAILED DESIGN - THE VPE GENERAL PROCEDURES.

4.1. Overview.

The VPE General Procedures are, as already mentioned, called by all VPE Output Programs. Their purpose is to make as large a part as possible of the code for these programs to be as independent from a particular lesson, and to make the VPE Output programs as simple as possible. The advantages of such a design are:

(a) The simplicity of the output program makes their generation, whether automatically or by hand, a relatively simple task.

(b) The output program produced should be more independent of the level of development of the general procedures, as their interface with the procedures is particularly simple.

(c) The general procedures make many of the decisions at run-time that the parser of the existing VPE system has to make at compile time, thus reducing the complexity of a parser for the proposed system.

As suggested by the hierarchy, these procedures can be split into 3 major concerns, which will be dealt with individually in the following sections. However, it is necessary first to describe the data structures used by the procedures, as these alone are central to the system design.

4.2. Data Structures For The General Procedures.

To understand the functions and the mechanisms of the VPE Data Structure Procedures, it is necessary to understand the data structures that drive them. These data structures are generally designed for convenience of the procedures that use them and are initialized at run time by the state table data file and, to a lesser extent, by the user. These data structures are slightly more generalized than need be for the immediate purposes of this project, as are the data structure procedures themselves, for the purpose of future extensions. Three data structures are relevant, namely the state table, the state stack, and the program status record.

4.2.1. The Program Status Data Structure.

This data structure is defined by the responses of the VPE user, either directly or indirectly. It comprises of a record giving the current visible execution status of a lesson. The relevant fields are:

i. querymode: a flag set true if the user has opted to execute lesson in query mode, rather than in automatic. The user is asked only once at beginning of lesson.

ii. forwardmode: a flag set true if VPE lesson is executing in forward direction. This flag may be modified by user upon the prompt to change control.

iii. maxdepth: an integer giving the maximum depth screen state the lesson may pause at upon execution. This is modified by user upon prompt to change control, but is never allowed to be negative.

iv. askcontrol: this flag is set when the user requests to change control and reset when user is prompted for control.

v. failure: this flag is set whenever lesson is in query mode and the user has failed to give a correct answer to a prompt for a variable's value. This flag is written into the file reporting the success of the lesson.

4.2.2. The State Table.

The state table is the most crucial data structure for the state driven system, as it contains all relevant information, static and dynamic, for execution of a VPE lesson. It comprises of several smaller data structures, which are described below:

a. Screen variable list.
Each variable/ condition in a VPE displayed program has an entry to a node of this list. The following fields a screen variable list node are as follows:

i. id: contains the string value of the identifier/ expression corresponding to the program variable/ condition.

ii. row: contains the vertical screen position of the box for screen variable; if no box exists, it is given an illegal (negative) value.

iii. col: contains the horizontal screen position of the box for the screen variable; if no box exists, it is given an illegal (negative) value.

iv. specialdisplay: a flag true if special box display effects (ie. box label in inverse video) required.

v. displayed: a flag true if the box of the screen variable is currently displayed on screen.

vi. vartype: a scalar variable signifying the base type (ie. integer, boolean, condition) of the screen variable.

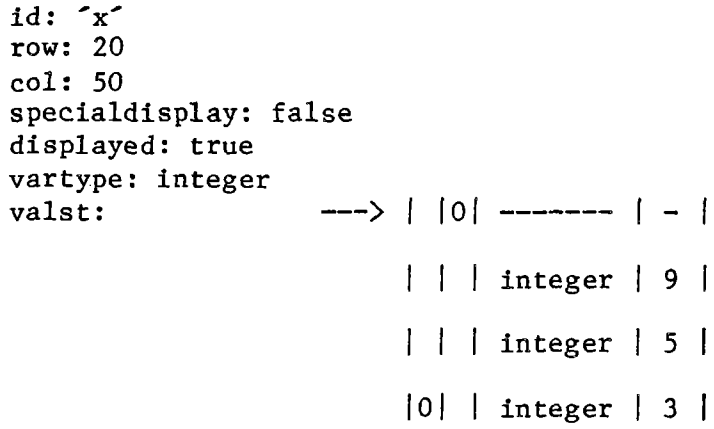
vii. isarray: a flag true if the screen variable is an array.

viii. valst: a variant field, defined on isarray false, pointing to top of a stack of values that have been assigned to the screen variable.

ix. valstarr: a variant field, defined on isarray true, pointing to an array of value stacks, described above, for each element of the array.

x. lob, hib: these are variant fields, defined on isarray true, giving the lower and upper bounds, respectively of the array corresponding to the screen variable.

Example:
An integer 'x' in a VPE displayed program is currently displayed at co-ordinates (20,50) on the screen and has values 3,5,9 assigned to it during visible execution. The corresponding screen variable list node is as follows:



b. Index List.
This is a list of indices (integer expressions) accessing the arrays in the VPE displayed program. An array index must be purely integer expressions, with unary operators '+', '-', and binary operators '+', '-', '*', 'div', The relevant fields of each node in an index list are as follows:

i. id: string value of the index.

ii. postfixid: string value of the postfix form of the index. note: unary '-' is represented by '_' in this convention, to avoid confusion.

iii. value: an integer giving the current value of the index; this is periodically calculated during the lesson by a postfix interpreting procedure.

iv. alist: a list of whose nodes contain a pointer to screen variable list array and the 'offset' of the index to that array.

Example: index 'i+j' indexes array 'a' (offset 2) and has current value 2 has following node in index list:

```
id: 'i+j'
postfixid: 'i j +'
value: 2
alist:              --> <ptr. to screen var. 'a'>
                            offset: 2
```

c. Scope Tree.
Defining a module to refer to a procedure, function or program in the VPE displayed program, a node in the scope tree contains relevant information to such a module. (the term 'scope tree' is borrowed from compiler design in which symbol tables are defined for each module). In the proposed system, there can only be one module in any VPE displayed program, but this data structure is left in a more general form for later expansion. A node in a scope tree contains the following fields:

i. modid: the string value of display program identifier for the module.

ii. varlist: a screen variable list of all parameters/ variables/ conditions local to module

iii. indexlist: a index list of all array indices local to module.

iv. statementcount: a dynamic field, counting the number of executable Pascal statements visibly executed so far during the lesson.

Example:
For the following VPE displayed program:

```
"procedure lesson (x: integer);
begin
        x := x + 1
end;"
```

assuming the lesson is at the stage where x is incremented, the corresponding scope tree node would be:

```
modid: 'lesson'
varlist:              --->      <screen var. list
indexlist : nil                   node for 'x'>
statementcount: 1
```

The state table references all these data structures, directly or indirectly. The state table is an array of all screen states of the displayed program, and each entry contains the following relevant fields:

i. modptr: a pointer to the scope tree node for the module containing the state.

ii. reldepth: depth of the state relative to the first state (entry to the module) of the corresponding module.

iii. textpos: gives the textual position, ie. the line number in the displayed program, of the code corresponding to the state.

iv. variablelist: a list of screen variable nodes affected by visual execution of that state.

v. statetype: a scalar field signifying the type of the screen state. ie. assignment statement, array element assignment, condition evaluation, etc.

vi. iptr: a variant field, defined on statetype being an array element assignment, pointing to the appropriate node in the index list for the array index appearing on the left of the assignment.

vii. whiteset: a variant field, defined on statetype being an array element assignment, true if the array element is to be left in inverse video after execution of the state.

Example:
In the above lesson, state 2 would correspond to the assignment

```
modptr:                  ---> <scope tree node
reldepth: 1                       as above>
textpos: 2
variablelist:            ---> <screen variable list
statetype: assignment.            node for 'x'>
```

4.2.3. The State Stack.

The state stack is a history of all screen states visibly executed during a lesson. It enables forward and reverse visible execution. It is represented as a dequeue of indexes into the state table. When the lesson is executing in reverse mode, the pointer to the top of the state stack is decremented; when mode is changed to forward again, the pointer is incremented until the top of the stack is reached.

4.3. The Data Structure Initialization Routines.

These routines initialize the program status data structure and the state table, and must be called before visible execution begins. The initialization for the state stack data structure is trivial. These routines are divided into two orthogonal concerns:

4.3.1. The Program Status Initialization Procedure.

This procedure is the first VPE Data Structure Procedure called. The algorithm is as follows:
- give user optional instructions and input querymode field from user.
- set forwardmode to true.
- set maxdepth to pause at all screen states.
- display the VPE Displayed Program.

4.3.2. The state table initialization procedures.

The state table for a VPE lesson is initialized by reading the state table file. The main advantages of initializing the state table in this way, as opposed to 'hard coding' of the state table information in the VPE Output program, are:

(a) the automatic generation of the VPE output programs is greatly simplified.

(b) the screen layout of a VPE lesson, ie. details of the display of a variable, can be modified without needing to recompile the VPE Output program.

Because of the complex nature of the state table, it is necessary to give a grammar defining how the state table data file must be structured. The procedures parse this grammar using a recursive descent scan, and build the state table appropriately. The grammar for the data file is given below semi-formally, but first the reader should note the following conventions used in this grammar:

i. Characters appearing in quotes are taken as literal characters.

ii. Symbols in lower case correspond to state table initialization procedures.

iii. Upper case symbols correspond to fields in the data structure created by the production. In the case of a field being a scalar variable, its ordinal value is expected in the data file. If the field is an integer or string, its textual value is expected.

iv. Semantic actions, which occur as soon as possible during productions, appear in braces underneath the appropriate production. These indicate how the data structure fields, not directly read from the data file, are initialized. References to symbols in the productions appear on the right of assignments to fields; (only symbols corresponding to procedures may be referenced this way), and gives the data structure that particular procedure returns.

v. Comments, enclosed within '(*' and '*)', describe the functional purpose of each production, and what, if anything, is returned.

Grammar for the state table data file:

```
(* getscopetree: builds a scope tree node,
      then reads in states for state table;
      getstates inherits a pointer to this
      node *)
getscopetree -> '&m\' MODID'\' '/n' getscreenlist
                getindexlist getstates 'm&\' '/n'

                { VARLIST <- getscreenlist
                  INDEXLIST <- getindexlist
                  STATEMENTCOUNT <- 0 }

(* getscreenlist: builds a screen variable
        list node; appends this to a list
        of following screen variables and
        returns pointer to head of list *)
getscreenlist -> '&v\' ID'\' ROW COL VARTYPE
                 ISARRAY '/n' getscreenlist

                 { assert ISARRAY is false
                   DISPLAYED <- false
                   append list returned by
                     call to getscreenlist to
                     screen  variable list node}
                 |
                 '&v\' ID'\' ROW COL VARTYPE
                 ISARRAY LOB HIB '/n'
                 getscreenlist

                 { assert ISARRAY is true
                   DISPLAYED <- false
                   append list returned by call
                     to getscreenlist to screen
                      variable list node }
                 |
                 'v&\' '/n'

                 {return nil list}


(* getindexlist: builds an index list node and
         appends a list of indices
         to it; returns pointer to head of
         list *)
getindexlist -> '&i\' ID'\' POSTFIXID'\'
                getarraylist '/n' getindexlist

                 {IVAL <- "undefined value"
                  ALIST <- $3 }
                 |
                 'i&\' '/n'
                 {return nil list}
```

```
(* getarraylist: reads in array list for
        an index list node *)
getarraylist -> getword OFFSET getarraylist

              {if $1 is not in screen
                  variable list of scope
                  tree then error
                else
                  SCRARRAY <- scope tree screen
                         variable lookup of $1}

               |
               "empty"

               {return nil array list}


(* getword: read in characters until '\'
          or newline encountered
          returns string value of characters *)
getword ->      "sequence of characters '\' or
              newline terminated"


(* getstates: reads next state into the next free
    element of the state table data structure *)
getstates ->    '&s\' RELDEPTH TEXTPOS STATETYP
                 getvarlist '/n' getstates

                { assert STATETYPE <>
                              element assignment
                  MODPTR <- inherited pointer to
                               scope tree node
                  VARLIST <- getvarlist
                  increment state counter
                }
                |
                '&s\' RELDEPTH TEXTPOS STATETYP
                getword WHITESET getvarlist

                { assert STATETYPE =
                              element assignment
                  MODPTR <- inherited pointer to
                               scope tree node
                  VARLIST <- getvarlist
                  IPTR <- scope tree index list
                              search of getword
                  increment state counter
                }
                |
                's&\' '/n'
```

The initialization procedures will 'panic' or give a Pascal input error if the state table data file is not on this grammar. The main state table initialization procedure returns the state table and its algorithm is as follows:

- set state counter to 0
- reset state table data file
- call getscopetree procedure
- assert state counter <> 0

4.4. The 'Universal' General Procedure.

This procedure is called by the VPE Output Program, for every screen state, in a manner described in the specifications. It has a defined effect once the data structures state table and program status are initialized. The idea behind this procedure is to simplify greatly the VPE Output Program, replacing typically several procedure calls with long parameter lists with a single, typically simple, procedure call. The procedure is the 'driver' for the general procedures; it is responsible for maintaining the data structures and for (indirectly) adjusting the screen. The procedure maintains a 'history' of the VPE lesson, and has a loop in which, once user has requested visible execution in reverse mode, the pointers to the state stack data structures and the appropriate value stacks of the screen variables are incremented/decremented until visible execution has reached the same point as when the loop was entered. The procedure has two arguments:

i. state number:
the index into the state table corresponding to the screen state for the code in the VPE Output Program that called the procedure.

ii. value list:
a list of values of the screen variables affected by execution of that state; the list is in the same order as the variable list in the corresponding entry into the state table.

The algorithm for the procedure is given below:

- push state number onto the state stack
- push the values in value list onto the appropriate screen variable value stacks
- reset switched mode flag
- screen adjust for forward mode: pass state number and switched mode flag
- while execution is not in forward mode or state stack pointer is not at top of state stack do
    - if execution in forward mode then
        - state number <- 'top' state given by state stack pointer
        - decrement value stacks of appropriate screen variables
        - screen adjust for reverse mode: pass state number and switched mode flag
        - decrement state stack pointer
    - else
        - increment state stack pointer
        - state number <- 'top' state given by state stack pointer
        - increment value stacks of appropriate screen variables
        - screen adjust for forward mode: pass state number and switched mode flag

4.4.1. The 'high level screen adjusting' procedures.

These procedures, called by the 'universal' general procedure, use the data structures state stack and program status to call the VPE Screen Procedures. On the top level are the 'screen adjust for forward mode' and 'screen adjust for reverse mode' procedures, called by the 'universal' general procedure. Both these procedures exhibit a degree of symmetry, and have, as arguments, the state number and a switched mode flag.
The algorithm for the 'screen adjust for forward mode' procedure is as follows:

- let scopemod be pointer to scope tree node corresponding to the state number
- adjust statementcount of scopemod
- if state is visible and not switched mode then
    - set arrow on screen, according to text position of state and statement counter
- if state is an entry to a module and not switched mode then
    - adjust boxes of scope module
- adjust screen variables, according to variable list of state
- if state could have affected array indexes of scopemod then
    - adjust array indexes of scopemod
- determine program control, according to program status (return switched mode flag)
- if state is visible and an exit from a module then
    - adjust boxes of scopemod
- if arrow for state needs to be deleted then
    - delete arrow, according to textual position of state
- if state is last state of the lesson then
    - tell of success of lesson

The algorithm for the 'screen adjust for reverse mode' procedure is as follows:

- let scopemod be pointer to scope tree node corresponding to the state number
- if state is visible and not switched mode then
    - set arrow on screen, according to text position of state and statement counter
- if state is an entry to a module and not switched mode then
    - adjust boxes of scope module
- adjust screen variables, according to variable list of state
- if state could have affected array indexes of scopemod then
    - adjust array indexes of scopemod
- adjust statementcount of scopemod
- determine program control, according to program status (return switched mode flag)
- if state is visible and an exit from a module then
    - adjust boxes of scopemod
- if arrow for state needs to be deleted then
    - delete arrow, according to textual position of state

4.4.1.1. Sub - algorithms.

The procedures called by the above screen adjusting procedures will be described, in functional terms in this section. All these procedures have access to the current state.

i. adjust boxes of scope module procedure.

- for each box in the screen variable list
    - if box must be deleted and box is displayed on screen then
        - delete box of screen variable
    - else if box must be set and box is currently displayable then
        - set box of screen variable

ii. adjust screen variables, according to variable list of state

- for each screen variable in list
    - if box for screen variable is displayed then
        - call appropriate 'getset' VPE Screen Procedure for screen variable (parameters are determined from the value stack, type, row, col of screen variable; and from the state and program status. In the case of array element assignment, the index value for the state is also needed)
    - else if the state is visible and has only singleton variable list then
        - send 'delay' to user
- if state is visible and a module exit then
    - send 'delay' to user

iii. adjust array indexes of a scope tree node.

- for each index list node in index list in scope tree node
    - if state is a module exit then
        - set value of index to be 'undefined'
    - else
        - postfix evaluate index according to the postfix string value of index
    - for each array in array list of index
        - position index under array, using old and new values of index, and its offset to that array.

In this procedure, a function that interprets the postfix string form of the index is used and returns an integer result. This routine uses the value stacks corresponding to the screen variables in the postfix string; if any of these value stacks are empty, an 'undefined' result is returned.

iv. determine program control, according to program status

- if user has requested to change control then
    - call get control response procedure
    - if user has asked for an allowed maximum state depth then
        - change maximum state depth to that value
- if current state is the lowest visible state on the state stack then
    - set forward mode
- else if user has asked for a particular mode then
    - alter forward mode flag accordingly
- reset the flag for the change control request.
- return whether mode has switched

v. adjust statementcount of scope tree node

- if state is entry to a module then
    - reset statement count
- else if state is executable then
    - increment/ decrement statement count, according to forward/ reverse mode

4.4.2. The data structure maintenance procedures.

These are also called by the 'universal' general procedure. The procedures for pushing/incrementing/decrementing the state stack are trivial, and algorithms for these will not be given in this section. The procedures for adjusting the value stacks of screen variables are more involved, particularly for the case of arrays. These procedures have the current state and its variable list as arguments and their algorithms are given below:

i. push list of values onto the appropriate screen variable value stacks

- for each screen variable in the screen variable list
    - let v be the corresponding value in value list
    - if screen variable is an array then
        - if the state is an element assignment then
            - let i be value of the index corresponding to the state
            - push the ith element of v onto the ith value stack of the screen variable
        - else
            - push each element of v onto the corresponding value stack of the screen variable
    - else
        - push v onto the state stack of the screen variable

ii. increment value stacks of appropriate screen variables

- for each screen variable in the screen variable list
    - if screen variable is an array then
        - if the state is an element assignment then
            - let i be value of the index corresponding to the state
            - increment the ith value stack of the screen variable
        - else
            - increment all value stacks of the screen variable array
    - else
        - increment the value stack of the screen variable

iii. decrement value stacks of appropriate screen variables

- for each screen variable in the screen variable list
    - if screen variable is an array then
        - if the state is an element assignment then
            - let i be value of the index corresponding to the state
            - decrement the ith value stack of the screen variable
        - else
            - decrement all value stacks of the screen variable array
    - else
        - decrement the value stack of the screen variable

4.5. The VPE Screen Procedures

These procedures directly manipulate the screen layout. On the existing system, the equivalent for procedures appear in the UNIX directory '/pub/vpe/explain/general' under UNIX file names such as 'general.i', 'array.i', etc. The procedures used on the state-driven system are largely similar to those mentioned above, except that their organization is made more consistent, algorithms modified and arrays are handled differently. These procedures are independent of the data structures used by the Data Structure Procedures previously described. They have the following functions:

i. to manipulate the screen to create the VPE effects.

ii. to request the VPE user for input and to to receive and verify the user's responses to these requests.

iii. to read from the lesson's data file (except the state table data file), and to write to the file reporting the success of a lesson.

iv. to return flags to the Data Structure Procedures on the user's responses.

The calling hierarchy of these procedures is given in section 1.2; on the highest level, these procedures are called from the Data Structure Procedures. It is assumed that the reader is familiar with the equivalent procedures in the existing system, and only the design of the procedures substantially modified is given in the following subsections.

4.5.1. Organization of the 'getset' procedures.

The 'getset' procedures refer to the procedures needed to get the user's response for input of a particular 'atomic' variable, and display the correct value on the screen. These procedures have been extended to meet extra requirements of the state driven system. A functional description of the 'getset' procedures needed for each 'atomic' data type is given below:

i. flash value routine.

- flash the textual value of a variable, with possibility of leaving in inverse video

ii. check user response routine.

- read in user's response
- return whether user's response was valid
- return the value of the user's response
- return flag signifying whether the user has requested for control

iii. get user's response routine.

- flash in inverse video, the old value of the variable, if it exists, each time before 'check user response routine' called
- give the user several attempts to give valid input to the check user's response routine.
- if user fails to do this, stay in an input loop until correct escape character entered
- return value of user's response
- return whether the user requested for control

iv. 'getset' user's response routine

- if visible execution is pausing at this step and value is valid then
    - if in query mode then
        - for a fixed number of chances, call the get user's responses routine and flash the user's response in normal video
        - if user has not given correct answer, set failure flag
    - else
        - 'delay', (returns whether user has asked for control)
    - flash correct value in box, with option of inverse video
- else
    - if visible execution pauses at this step then
        - 'delay', (returns whether user has asked for control)
    - delete existing value of variable on screen
- return flag whether user has failed
- return flag whether user has asked for control.

4.5.2. Handling of arrays.

With the 'getset' procedures handled as above, and the changes in the user interface of the VPE Screen Procedures with respect to arrays, the handling of arrays by these procedures is particularly simple. The extra routines required for arrays are independent of the routine to set an array box (of variable length, but each element box has fixed width), delete an array box, and position the index under an array. Hence, to 'getset' an array element of type integer, say, a call to the 'getset' integer routine would be required, passing the old and new values of the appropriate array element, the appropriate screen co-ordinates of the array element, and a flag determining whether to set element in inverse video. Also, to set a range of array elements (only done on a module entry state which an array is a parameter), the 'getset' routine is called iteratively, without pausing.

4.5.3. The get user control response routine.

This routine, whose effect has been described in section 1.4, has the following functional description:

- scan the user's response
- return the maximum depth of state for execution to pause at
- return the execution mode

5. THE DESIGN OF THE VPE PARSER.

The VPE Parser is implemented as a Llama compiler. The advantages of this are as follows:

a. built in syntactic input error checking.

b. built in trace facility.

c. source code is more readable, as code for semantic actions is not interleaved with code for syntactic analysis (as in a recursive descent compiler),

d. source code is generally easier to extend.

A disadvantage of having a Llama compiler is that the object code program, for the existing Pascal software on the University of Wollongong UNIX system, is very large. The VPE Parser will assume the VPE displayed program is semantically correct, as little semantic error checking will be done by the parser. The compiler will only parse the subset of Pascal VPE Displayed program, as described in the next section. The design of the VPE Parser is, as already stated, is considerably simpler than the parser for the existing system, for the following reasons:

(a) the organization of the data structures require no large lesson-dependent constant declarations.

(b) the simplified interface of the VPE Output programs with the VPE General Procedures allows considerably simpler code generation for the Parser.

(c) the parser is concerned with two largely orthogonal tasks, namely production of the state table data file and the code of the output program, rather than one larger task.

(d) the parser can produce the state table data file for a lesson after completion of parsing the displayed program; hence, final decisions for production of this file can be deferred until the parsing is complete, and all relevant information has been gathered.

5.1. Acceptable Displayed Programs For The VPE Parser.

The parser will use a LR Pascal grammar, which accepts valid displayed programs according to the current development of the VPE General Procedures. As well as the restrictions of the displayed program mentioned for the specifications for this project, the further restrictions will hold:

i. the displayed program must have no internal declarations of procedures and functions. (for the current level of sophistication of the general procedures).

ii. the displayed program must have no Pascal input / output statements.

iii. the displayed program must have no declarations of, or references to, a non-standard VPE type.

5.2. Handling of input errors.

The following two types of input errors can occur:

i. Syntactic error.
In this case, the Llama compiler will abort, without explanation. The trace facility must be used to find the nature of the syntax error.

ii Semantic error.
Here, the error may be detected, (if it is of type iii. in above section), in which case a diagnostic will appear to terminal, and the parser will halt. However, usually the semantic error will be ignored by the parser, and will be detected upon compilation of the VPE Output Program produced by the parse.

5.3. Tasks Of The VPE Parser.

The Parser will perform the following two tasks in parallel:

i. Build the state table for the VPE Output Program.
The state table will be built up during the parse into a similarly defined state table as defined for the VPE Data Structure Procedures. Most information for the state table is deduced from the input program, but the parser will ask the VPE Preparer to enter some information, from standard input, on:

(a) upper and lower bounds of array variables.

(b) whether to 'white set' an array element, upon an array element assignment.

(c) how array indices are to be displayed. (ie. their offset underneath the array boxes.)

Before the parse begins, the parser will read the output file of the ISP program, store the information in records; it merges this information into the state table at the end of the parse.

ii. Generate code for the VPE Output Program.
The parser will generate sequential code for the VPE Output Program. Most information to generate the code can be determined from the input program, but the following must be separately determined:

(a) The definitions of the constants SAFEROW, SAFECOL. These are determined from the ISP output file.

(b) The initialization of displayed program parameters. These must be supplied by the preparer. For integer parameters, the preparer will firstly be asked whether the value is to be randomly generated. If so, the preparer will be requested the range value can be in, otherwise the preparer supplies the exact value. For boolean parameters, the preparer is asked to supply initial value. For integer arrays, the values are randomly generated, and preparer is asked to supply range. The parser will the output the appropriate initialization procedure calls to the output program.

The details of code generation are given in the following section.

5.4. Code Generation.

As the form of the VPE Output Program closely reflects that of the corresponding displayed program, the code generation chiefly consists of two processes:

i. Input code reflection.
This occurs for the following displayed program constructs:

(a) declarations of individual constants and variables. (type declarations ignored)

(b) begin, repeat and end keywords.

(c) assignment statements.

(d) code for the head of while, if and for statements.

Repeat statements and program declarations are modified, as shown in the following subsections.

ii. Code insertions.
Various types of code must be inserted between the 'reflected' code to make up a VPE lesson:

(a) #include directives for definitions of the constants, types, global variables and the VPE General Procedures.

(b) definitions of the constants 'SAFEROW', 'SAFECOL'.

(c) parameter initialization statements.

(d) VPE data structure initialization procedures calls. (called 'Vinitprog',

(e) calls to 'universal' general procedure and necessary begin and end keywords.

The principal attribute of the symbols in the grammar is a list of characters representing the textual value of the symbol. This is particularly convenient for code 'reflection'.

5.4.1. Code generation for Pascal Displayed program constructs.

The code generating actions of the VPE Parser will be informally defined in this section. An example of a part of a VPE Displayed program is given, in pseudo code, and the corresponding parser output is written underneath, again in pseudo code.

i. Procedure Declaration.
Displayed program code:

```
"procedure x ( <parameter list of x> );
const
        <constant declarations of x>
type
        <type declarations of x>
var
        <variable declarations of x>
begin
        <statements of x>
end;"
```

Parser Output:

```
"program VPE (input, output);
#include 'Vconst.i'
        <constant declarations of x>
#include 'Vtype.i'
#include 'Vvar.i'
        <var. declarations for parms. of x>
        <variable declarations of x>
#include 'Vproc.i'
begin
        <initialization code for parms, of x>
        Vinitprogstatus (Vprogstatus, 'x');
        Vinitstatetable (Vstatetable, 'x.s');
        Vgeneral (1 , <value list containing
             values of parameters of x>);
        <VPE output code for statements of x>
        Vgeneral (<last state>, nil);
end."
```

ii. Program Declaration.
Displayed program code:

```
"program x ( <file list of x> );
const
        <constant declarations of x>
type
        <type declarations of x>
var
        <variable declarations of x>
begin
        <statements of x>
end."
```

Parser Output:

```
"program VPE (input, output);
#include 'Vconst.i'
        <constant declarations of x>
#include 'Vtype.i'
#include 'Vvar.i'
        <variable declarations of x>
#include 'Vproc.i'
begin
        Vinitprogstatus (Vprogstatus, 'x');
        Vinitstatetable (Vstatetable, 'x.s');
        Vgeneral (1 , nil);
        <VPE output code for statements of x>
        Vgeneral (<last state>, nil);
end."
```

iii. assignment statement.
Displayed Program code:

```
"x := <exprn>"
```

Parser output:

```
"x := <exprn>;
Vgeneral (<current state number>,
         <list containing value of x>);"
```

iv. array element assignment statement.
Displayed Program code:

```
"x[<exprn1>] := <exprn2>"
```

Parser output:

```
"x[<exprn1>] := <exprn2>
Vgeneral (<current state number>,
   <list containing value of x>);"
```

v. compound statement:
Displayed Program code:

```
"begin
<statement list>
end"
```

Parser output:

```
"begin
<Parser output code for statement list>
end"
```

vi. while statement.
Displayed Program code:

```
"while <exprn> do
        <statement>"
```

Parser output:

```
"while <exprn> do
        begin
        Vgeneral (<state number of while loop>,
                valcond (true));
        <parser output for statement>
        end;
Vgeneral (<state number of while loop>,
   valcond (false))"
```

vii. if statement.
Displayed Program code:

```
"if <exprn> then
        <statement>"
```

Parser output:

```
"if <exprn> then
        begin
        Vgeneral (<state number of if stmt.>,
                valcond (true));
        <parser output for statement>
        end
else
        Vgeneral (<state number of if stmt.>,
                valcond (false))"
```

vii. if-then-else statement.
Displayed Program code:

```
"if <exprn> then
        <statement1>
else
        <statement2>"
```

Parser output:

```
"if <exprn> then
        begin
        Vgeneral (<state number of if stmt.>,
                valcond (true));
        <parser output for statement1>
        end
else
        begin
        Vgeneral (<state number of if stmt.>,
                valcond (false))"
        <parser output for statement2>
        end"
```

viii. Repeat statement.
Display program code:

```
"repeat
        <statement list>
until <exprn>"
```

Parser Output:

```
"repeat
        <parser output for statement list>;
        Vtmpbool := <exprn>;
        Vgeneral (<state number of repeat stmt.>,
                        valcond (Vtmpbool))
until Vtmpbool"
```

ix. For statement.
Display program code:

```
"for x := <exprn1> to <exprn2> do
        <statement>"
```

Parser Output:

```
"for x := <exprn1> to <exprn2> do
        begin
        Vgeneral (<state number of for loop>,
                valint (x));
        <parser output for statement>
        end;
Vgeneral (<state number of for loop>,
         valint (x));   "
```

6. PROJECT IMPLEMENTATION.

The project implementation includes the implementation of the VPE General Procedures and the preparation of several VPE lessons. The coding is in Pascal, and due to the limitation of Pascal object file size on the current UNIX system, algorithmic clarity has been less priority than concise code. A description of the UNIX file names of the general procedure and lesson files is given in a file named 'READ ME' left in the main directory of the project files.

6.1. Implementation Of The VPE Data Structure Procedures.

The Data Structure Procedures are implemented in the hierarchy given in section 3.2.1 ii. The implementation takes the form of Pascal #include files, for data structure constant, type, variable and procedure definitions, respectively. These four files, together with the #include file for the VPE Screen Procedures, contain 3300 lines of documented Pascal code, only slightly larger than the library procedures for handling VPE integers, conditions, arrays and characters in the existing VPE system. In the VPE output programs, these files are explicitly referenced in #include directives, and are compiled with the output program.

These procedures all have error checking facilities; in particular errors from bad state table data file format are checked. These procedures have error tracing code, in which information is written to a debug file in the current directory at run time, except the data structure primitive procedures. For their interface with the VPE preparer, the reader is referred to section 3.4.ii.

All files referenced by these procedures, ie. the displayed program file, the displayed program state table data file, the lesson result file and the debug file, are, in the current implementation, assumed to be in the current directory. Hence, the output program code will need to be modified for execution in other directories. (ie. modified to include the full UNIX pathname for each file).

6.2. Implementation Of The VPE Screen Procedures.

These procedures are implemented in the form of a Pascal #include file, and a #include directive appears at the beginning of the file containing the VPE data structure procedure library. These procedures have a slightly different VPE user interface than the general procedures of the existing system, as described in section 3.4.i. Many of the algorithms used for the screen procedures are similar to the corresponding algorithms found in the general procedures for the existing system, but have been made more concise (in particular the 'getset' procedures; see section 4.5), and more flexible, to meet the needs of the state-driven system. Due to the modified user interface, and to the screen layout modification procedures being less complicated than those for the existing system, the overall code of these procedures is greatly reduced from the general procedures of the existing system.

6.3. Implementation Of VPE Output Programs.

Several lessons have been prepared to demonstrate the design feasibility of the state-driven system. Those prepared include VPE lessons for a while loop demonstrating program, a selection sort procedure, an integer array merge procedure, a function demonstrating the josephus problem, a binary search procedure, a string encryption procedure, and a boolean addition demonstrating procedure.

Each of the above lessons has a displayed program, state table data file, and an output program; the state table data file and the output program are in a format as though they were automatically generated. The state table data files are implemented as record files, suitable for easy automatic generation and scanning. The format of these files in given grammatically in section 4.3.1. The UNIX file name of a state table data file is, by the convention used by this project, the file name of the corresponding displayed program, ended in '.s'.

The output programs are implemented as Pascal programs, with #include directives to the VPE Data Structure library files, assumed to be in current directory. Most of these programs initialize their variables using time-dependent random data generation techniques. The output programs have code as outlined in section 5.4. The UNIX file name of an output program file is, by the convention used by this project, the file name of the corresponding displayed program, ended in '.p'.

The above lessons test most cases in the current implementation of the state-driven system. They are useful references for the VPE preparer when preparing future lessons.

7. PREPARATION OF LESSONS FOR THE STATE DRIVEN SYSTEM.

The following is a guide for the VPE Preparer to prepare a lesson for the state-driven system. Assuming the preparer has a displayed program called 'lesson', suitable for the state-driven system, the following steps must be undertaken:

i. decide the upper and lower bounds for any arrays in the displayed program, which variables and array indexes should be displayed on the screen, and, for each array element assignments, whether the array element should be left in inverse video.

ii. execute an ISP program [1] to assist the preparer in determining the screen layout for the displayed program. The preparer should note, preferably by hand:

(a) the row and column co-ordinates of the boxes of screen variable, and whether the label is in inverse video. In the case of conditions, the preparer should note that no two conditions are displayed at the same time on the state-driven system, and that the column co-ordinates of booleans/conditions given by ISP should be incremented by 1 when this information is later put in the state table.

(b) the offsets underneath the array boxes of the corresponding array indices; the ISP program currently does not resolve these offsets, and hence the preparer must consider the position of array indices when determining the overall screen layout. The offsets must be at least 2 (ie. at least 2 lines the row co-ordinate of the array) to be displayed, and two indices of a given array may have the same offset, providing they cannot clash during execution of the lesson.

Note that for array boxes, individual boxes must have length 6. This step typically takes 10-15 minutes, depending on the lesson.

iii. decompose the displayed program into states.
(refer to the definition of screen states in section 1.2, and appendix A). This step typically takes about 5 minutes.

iv. prepare the state table data file for the lesson. This file has a grammar given in section 4.3.2.1; the meaning of each entry, in terms of the data structures of the general procedures, is also indicated in that section. The process of writing the file can be broken down in the following steps:

(a) enter the 'heading', namely:
"&m\ <name of displayed program>\"

(b) enter the screen variable information.
For each variable/condition of the displayed program enter across a new line:
"&v\ <identifier> <row> <column> <ordinal value of special display flag> <ordinal value of corresponding screen variable type> <ordinal value of the isarray flag>"
For array variables, lower and higher bounds should be given immediately following the above. The variable list ending line:
"v&\"
should now be entered.

(c) enter the index information.
For each array index, enter across a new line:
"&i\ <textual value>\ <postfix value of index>\ <list of array identifiers and corresponding offsets for index>"
Note that an offset value of less than 2 will not allow the index to be displayed. The index list ending line:
"i&\"
should now be entered.

(d) enter the state information.
For each state, enter, in the correct state enumeration order, across a new line:
"&s\ <depth of state> <line number in displayed program of state> <ordinal value of state type> <list of identifiers affected by that state>"
or, for array element assignments only:
"&s\ <depth of state> <line number in displayed program of state> <ordinal value of state type> <index identifier for that state> <ordinal value of element whiteset flag> <list of identifiers affected by that state>"
The state table ending line:
"s&\"
should now be entered.

(e) enter the data file footer, namely:
"m&\"

Note that in the above description, to find the ordinal values of VPE scalar types, the definitions of the corresponding scalar types should be consulted in the VPE Data Structure type definition file. Note also that all identifiers in the data file must be backslash terminated. An example of a state table data file is given in appendix A. An error in the state table format will result in the corresponding output program panicking at run-time; an incorrect value of a field is more difficult to detect. This step is the most difficult and error-prone when done by hand, but typically takes 5-10 minutes once the preparer is familiar with the data file format.

iv. prepare the VPE Output program.
The preparer must refer to the variable lists of the states given in the state table data file and the output program code generation, given in section 5.4. For the random initialization of displayed program parameters, the preparer should refer to the lessons prepared for this project, to call the appropriate random variable generation routines. The preparer is recommended to refer to a similar VPE output program already prepared, begin with a copy of the displayed program, and insert the appropriate VPE code. This typically takes 5-10 minutes to write and is less difficult than writing the state table data file; the output program should be less than fifty lines of Pascal code. The output program should be compiled with the '-wp' option.

8. EXTENSIONS FOR THE STATE-DRIVEN SYSTEM.

8.1. Addition Of A New Data Type.

To add a new data type, assuming the boxes for variables of this type are of standard length (six spaces), the VPE preparer must:

i. add a suitable screen variable constant identifier to the scalar type 'scrtype', (this must be added onto the end of the definition, otherwise the state table data files of prepared lessons may be corrupted), and perform appropriate change to the record variant in data structure 'valnode' found in VPE data structure definition file.

ii. design and code 'getset' procedures for that data type, following the algorithms given in section 4.5.2, and the existing 'getset' procedures for the state-driven system; these must be inserted into the VPE Screen Procedure library file.

iii. add the calls to these 'getset' procedures to the 'scrtype' case statement in VPE Data Structure procedure 'varadjust'.

iv. add singleton value list constructing primitives for the data type and the array of the data type.

v. update the 'scrtype' case statement of VPE Data Structure function 'valelement'

vi. add the value stack value retrieving VPE Data Structure function for the data type.

The system will now be able to handle examples of displayed programs containing variables of this data type, or of an array of this data type.

8.2. Addition Of Internal Procedures/ Functions.

To extend the system to handle displayed program with references to internally declared (non-recursive) procedures or functions, the following modifications are suggested:

i. Insert, into the screen state types definition, constants signifying the screen state types for procedure calls and function calls; a procedure call is 'executable', although a function call is not. Each such state has a screen variable list attribute of the variables passed by reference to that procedure.

ii. In the code of the VPE output program, insert an extra argument to the parameter list of each procedure/function, being the state number of the corresponding caller state.

iii. In the output program code for that procedure/ function, before the call to the 'universal' general procedure for the 'module' entry state, a 'linking' procedure must be called. This procedure takes as arguments the state number for the 'module' entry state and the state number for the caller state. This procedure 'links' the screen variables corresponding to the variables they are referencing. Henceforth, references to these 'screen' parameters by the data structure procedures will effectively access the corresponding 'global' screen variables. The procedure will also affect the depth of all states in the procedure function. ie. state depth will be the state depth of the caller state plus the relative depth of the state to the depth of the "modules'" entry state.

The state table initialization routines and the procedures called by the 'universal' general procedures are expected to need little modification for the inclusion of internal procedures/ functions.

9. CONCLUSIONS.

It is to be hoped that this project will serve as a basis for a more versatile and complete VPE system, due to its design advantages. The project is an illustration of how data structure design plays a crucial role in overall system design. Overall, the project has been chiefly design oriented. The following problems exist with the state-driven system:

i. The handling of array indices.
In the state-driven system, array indices have a very restricted form, due to the fact that indices need to be periodically evaluated during a lesson. The evaluation routine must be considerably modified to handle general expressions of type integer.

ii. The size of object code for the VPE output programs.
The size is typically large (presently approximately 30K), even for the output programs of trivial displayed programs. This is due to the fact that all the VPE General Procedures for the state-driven system must be compiled with the output program; these account for 95% of the object code.

A suggestion for the future design of the VPE Parser is to have the parser independent of an ISP program, and have it output the state table data file with the fields concerning screen layout unresolved. Then, an ISP program can be modified to accept this file as input, and query the preparer on screen layout accordingly. The advantages of such a design are as follows:

(a) the ISP program needs to ask fewer questions to perform the same task, and has ability to cope for variable scope (in lessons involving internal procedure/ function declarations).

(b) all questioning on screen screen layout can be done by one program, and is no longer a concern of the VPE Parser.

ACKNOWLEDGEMENTS.

I would like to thank my project supervisor Dr. R. G. Dromey, who was the supervisor for this project, for the good advice he has given to this project and the many hours he has spent assisting me.

BIBLIOGRAPHY.

[0]: DROMEY, R.G. 'Before Programming - On Teaching Introductory Computing'. Technical report, the University of Wollongong, 1981.

[1]: HILL, B.H. 'Interactive Screen Preparation For Visible Program Execution'. Technical report, the University of Wollongong, 1982.

APPENDIX A: EXAMPLE OF A LESSON - THE SELECTION SORT PROCEDURE.

The states of a selection sort is given below, inserted in braces in the selection sort procedure displayed program code. The depth of each state is also given, after the state number.

```
procedure selectionsort(var a:nelements; n:integer);
var i  ,  j  ,
    p  ,  min :integer;
begin {1, 0}
for {2,2} i := 1 to n-1 do begin
   {3,2} min := a[i];
   {4,2} p := i;
   for {5,3} j := i+1 to n do
      if {6,4} a[j] < min then begin
         {7,4} min := a[j];
         {8,4} p := j
         end;
   {9,2} a[p] := a[i];
   {10,2} a[i] := min
   end
{11,0} end
```

The VPE Output program for the selection sort procedure is:

```
program VPE(input, output);
#include 'Vconst.i'
  SAFEROW = 22;
  SAFECOL = 10;
#include 'Vtype.i'
#include 'Vvar.i'
    a           : nelements;
    n           : integer;
    i, j, p, min: integer;
#include 'Vproc.i'

begin
Vinitprog ('selectsort', Vprogstatus);
Vinitstatetable ('selectsort.s', Vstatetable);

n := 6;
Vtmpint := timeseed;
makerandomdata (1, 9, 1, 6, Vtmpint, a);

Vgeneral (1, valapp (valnel (a), valint(n)));
for i := 1 to n-1 do
   begin
   Vgeneral (2, valint (i));
   min := a[i];
   Vgeneral (3, valint (min));
   p := i;
   Vgeneral (4, valint (p));
   for j := i+1 to n do
      begin
      Vgeneral (5, valint (j));
      if a[j] < min then
         begin
         Vgeneral (6, valcond (true));
         min := a[j];
         Vgeneral (7, valint (min));
         p := j;
         Vgeneral (8, valint (p));
         end
       else
         Vgeneral (6, valcond (false));
     end;
     Vgeneral (5, valint (j));
   a[p] := a[i];
   Vgeneral (9, valnel (a));
   a[i] := min;
   Vgeneral (10, valnel (a));
   end;
  Vgeneral (2, valint (i));
Vgeneral (11, nil);
end.
```

The state table data file for the selection sort procedure is:

```
&m\ selectsort\
&v\     a\      18      9   0   4   1   1  6
&v\     n\      14      52  0   0   0
&v\     i\      5       52  0   0   0
&v\     j\      8       52  0   0   0
&v\     min\    14      71  0   0   0
&v\     p\      11      52  0   0   0
&v\     a[j] < min\ 22  70  0   1   0
v&\
&i\     i\      i\      a\      2
&i\     j\      j\      a\      2
&i\     p\      p\      a\      3
i&\
&s\     0       3       0       a\ n\
&s\     2       4       3       i\
&s\     2       5       1       min\
&s\     2       6       1       p\
&s\     3       7       3       j\
&s\     3       8       2       a[j] < min\
&s\     4       9       1       min\
&s\     4       10      1       p\
&s\     2       12      4       p\  0       a\
&s\     2       13      4       i\  1       a\
&s\     0       15      5
s&\
m&\
```